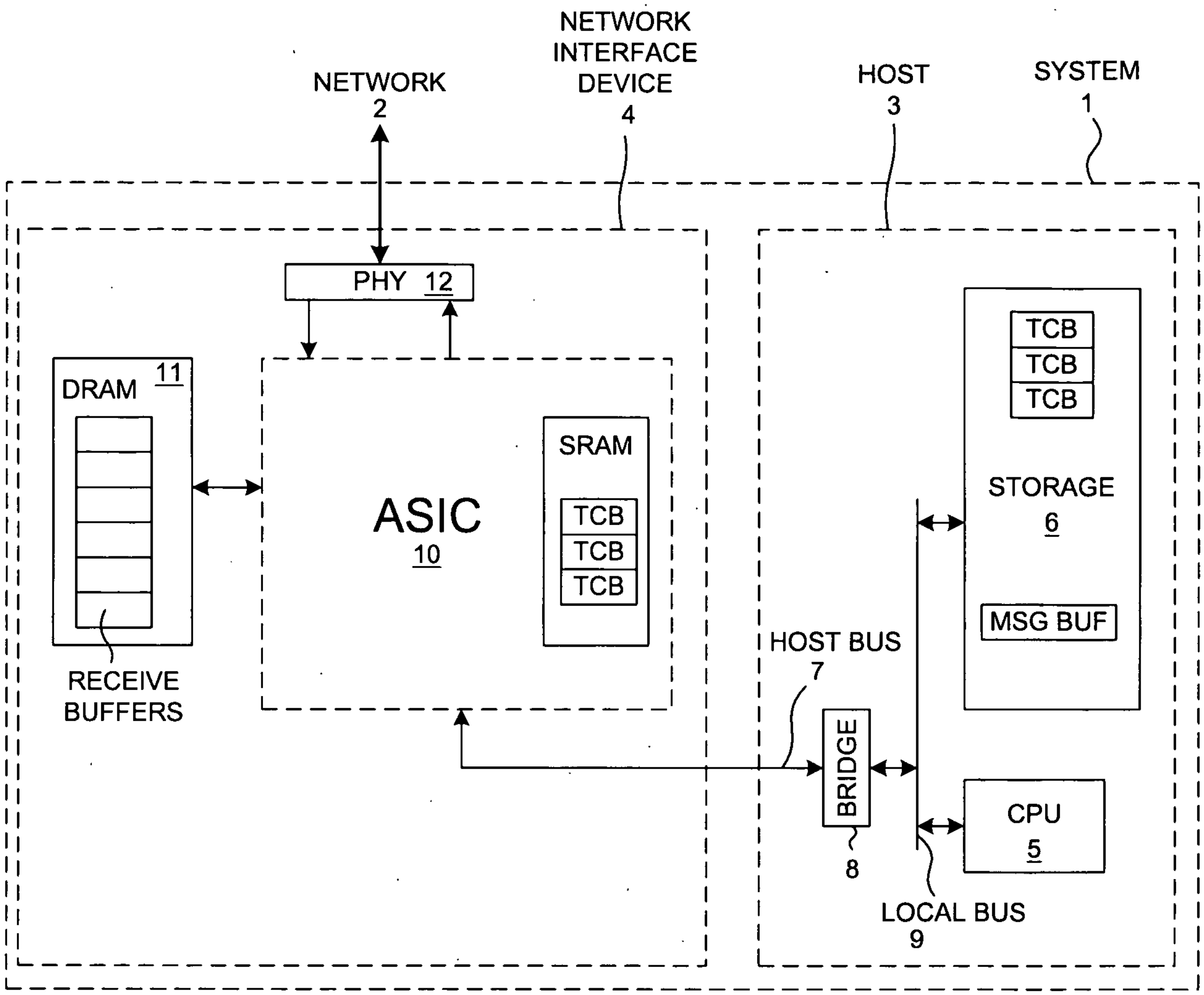

NETWORK
2
NETWORK
INTERFACE
DEVICE
4
HOST
3
SYSTEM
1
PHY 12
DRAM 11
ASIC
10
SRAM
TCB
TCB
TCB
RECEIVE
BUFFERS
TCB
TCB
TCB
STORAGE
6
MSG BUF
HOST BUS
7
BRIDGE
8
CPU
5
LOCAL BUS
9

FIG. 1

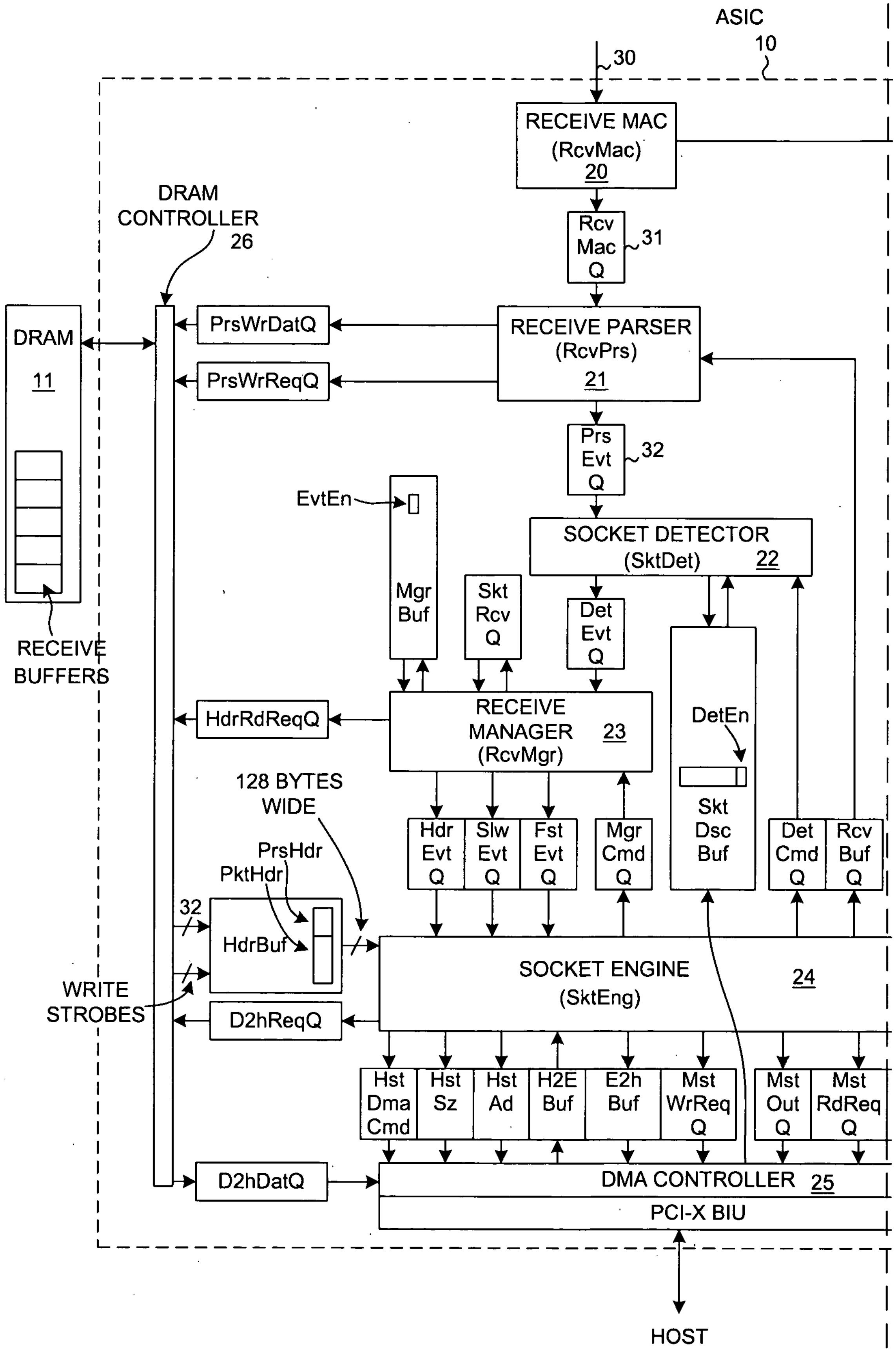

ASIC
10
30
RECEIVE MAC
(RcvMac)
20
Rcv
Mac
Q
31
DRAM
CONTROLLER
26
DRAM
11
RECEIVE
BUFFERS
PrsWrDatQ
PrsWrReqQ
RECEIVE PARSER
(RcvPrs)
21
Prs
Evt
Q
32
EvtEn
Mgr
Buf
Skt
Rcv
Q
SOCKET DETECTOR
(SktDet)
22
Det
Evt
Q
DetEn
Skt
Dsc
Buf
HdrRdReqQ
RECEIVE
MANAGER
23
(RcvMgr)
128 BYTES
WIDE
PrsHdr
PktHdr
32
HdrBuf
Hdr
Evt
Q
Slw
Evt
Q
Fst
Evt
Q
Mgr
Cmd
Q
Det
Cmd
Q
Rcv
Buf
Q
WRITE
STROBES
SOCKET ENGINE
(SktEng)
24
D2hReqQ
Hst
Dma
Cmd
Hst
Sz
Hst
Ad
H2E
Buf
E2h
Buf
Mst
WrReq
Q
Mst
Out
Q
Mst
RdReq
Q
D2hDatQ
DMA CONTROLLER
25
PCI-X BIU
HOST

FIG. 2A

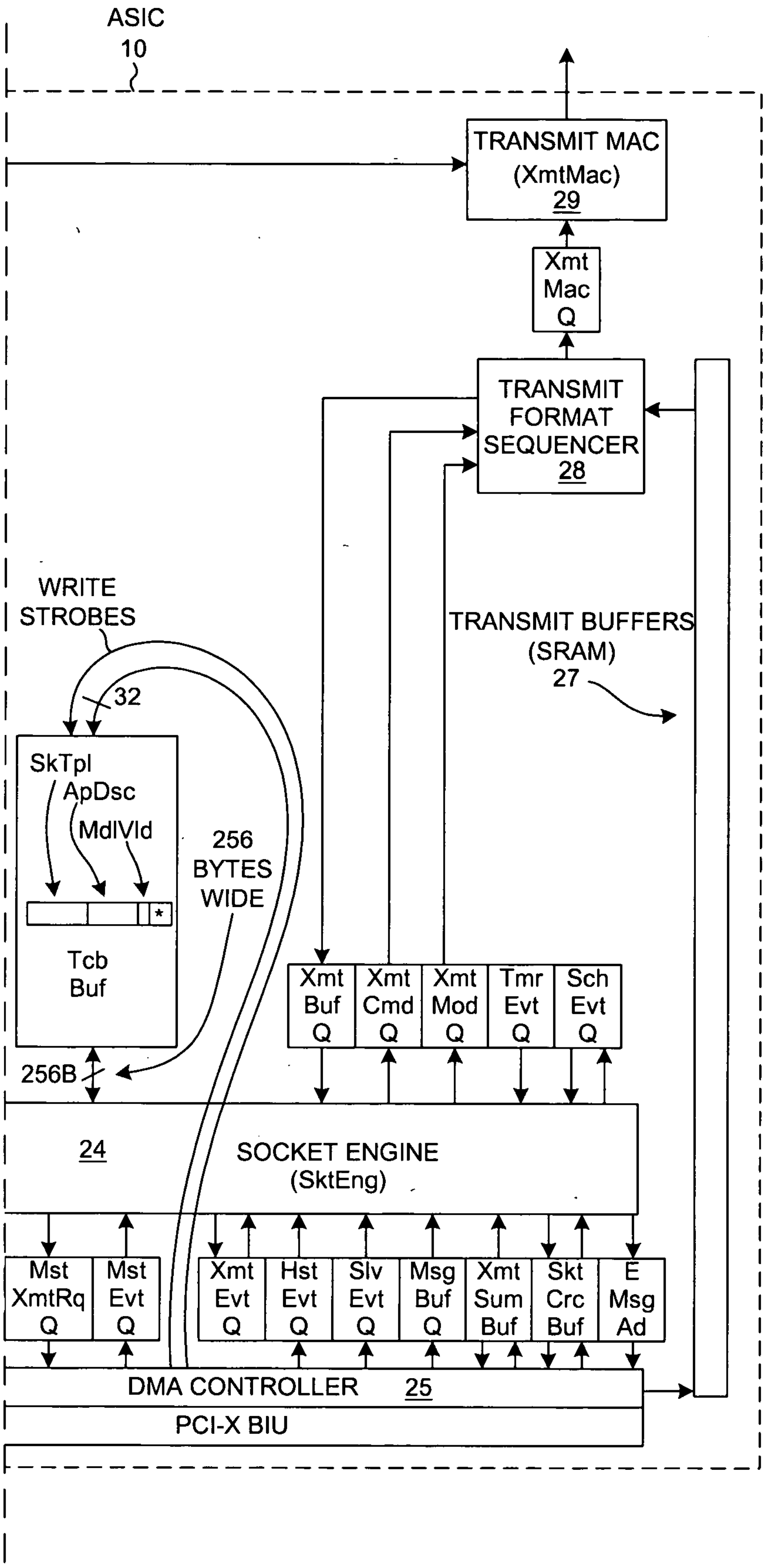


FIG. 2B

KEY TO FIG. 2

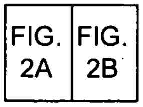


FIG. 2

BLOCK	DESCRIPTION
RcvMac	RECEIVE MEDIA ACCESS CONTROLLER
RcvPrs	RECEIVE PARSER.
SktDet	SOCKET DETECTOR.
RcvMgr	RECEIVE MANAGER.
SktEng	SOCKET ENGINE.

FIG. 3

MEMORY	FORMAT	DESCRIPTION
HdrBuf	[HdrId]{ PrsHd , PktHd }	HEADER BUFFER.
RcvMac	[BufId]{ PrsHd , RcvPk }	RECEIVE PACKET BUFFER.
SktDscBuf	[TcbId]{ SktDsc , DetEn }	SOCKET DESCRIPTOR BUFFER.
MgrBuf	[TcbId]{ EvtEn }	RECEIVE MANAGEMENT CONTROL.
TcbBuf	[TcbId]{ SkTpl , MdlVd , ApDsc , * }	TCB BUFFER (ASTERISK INDICATES ADDITIONAL VALUES - SEE FIG. 20)
HstMem	[MsgAd]{ MsgHd , MsgDt }	HOST MEMORY.
	[CmdAd]{ CmdHd , CmdDt }	
	[TcbAd]{ SktDsc , SkTpl , SktHdr }	
	[ApDsc]{ Data }	
RcvBufQ	{BufId }	QUEUE OF FREE DRAM RECEIVE BUFFERS.
RcvMacQ	{RcvPk , RcvSt }	RECEIVE PACKET BUFFERING QUEUE.
PrsEvtQ	{PkEnd , SkHsh , SktDsc }	PARSE EVENT QUEUE.
DetCmdQ	{EnbCd , TcbId }	SOCKET DETECTOR COMMAND QUEUE.
	{DblCd , TcbId }	
DetEvtQ	{EnbCd , TcbId }	SOCKET DETECTOR EVENT QUEUE.
	{DblCd , TcbId }	
	{SlwCd , HdrCd , PkEnd }	
	{FstCd , HdrCd , PkEnd , TcbId }	
MgrCmdQ	{ArmCd , TcbId }	RECEIVE MANAGER COMMAND QUEUE.
	{ClrCd , TcbId }	
	{PshCd , TcbId }	
	{ReqCd , TcbId }	
SktRcvQ	{HdrCd , PkEnd , HdrCd , PkEnd }	RECEIVE DESCRIPTOR QUEUES (0NE PER TCB).
SlwEvtQ	{EnbCd , TcbId }	SLOW RECEIVE QUEUE.
	{DblCd , TcbId }	
	{SlwCd , HdrCd , PkEnd }	
FstEvtQ	{ClrCd , TcbId }	FAST RECEIVE QUEUE.
	{FstCd , TcbId , HdrId }	
HdrEvtQ	{TcbId }	NOTIFY SKT ENG OF HEADER IN HDR BUFFER.
HstEvtQ	{CmdAd , TcbId }	SOCKET ENGINE COMMAND QUEUE.
MsgBufQ	{MsgAd }	QUEUE OF FREE HOST MESSAGE BUFFERS.

FIG. 4

MNEMONIC	DESCRIPTION
EngCx	SOCKET ENGINE CONTEXT.
HdSel	CONTEXT HEADER BUFFER SELECT.
TcbId	TCB BUFFER IDENTIFIER.
BufId	RECEIVE BUFFER IDENTIFIER.
HdrId	HEADER BUFFER IDENTIFIER.
CmdAd	HOST COMMAND ADDRESS.
MsgAd	HOST MESSAGE ADDRESS.
TcbAd	ADDRESS OF TCB BUFFER ON THE HOST.
ApDsc	APPLICATION BUFFER ADDRESS (STORED IN TCB BUF).
MsgHd	MESSAGE HEADER (MESSAGES ARE SENT FROM NID TO HOST)(INCLUDES A TCBID).
CmdHd	COMMAND HEADER (COMMANDS ARE SENT FROM HOST TO NID).
MsgDt	MESSAGE DATA.
CmdDt	COMMAND DATA.
SkHdr	PROTOTYPE TRANSMIT HEADER.
LnkHd	LINK HEADER.
SnpHd	802.3 SNAP HEADER.
VldHd	VLAN HEADER.
NetHd	NETWORK HEADER
TptHd	TRANSPORT HEADER.
PayLd	PACKET PAYLOAD.
PktHd	PACKET HEADER.
RcvSt	RECEIVE MAC PACKET STATUS (GENERATED BY RECEIVE MAC).
NetIx	NETWORK HEADER START INDEX.
TptIx	TRANSPORT HEADER START INDEX.
DdpIx	DDP HEADER START INDEX.
PkLen	PACKET LENGTH.
PkEnd	RECEIVE BUFFER ENDING ADDRESS.
SkHsh	SOCKET HASH {TCB[N-1:03]}
PrsSt	PACKET PARSE STATUS.
DetEn	SOCKET DETECTION ENABLE BIT.
MdlVd	MEMORY DESCRIPTOR VALID (STORED IN TCB BUF).
HdrCd	HEADER LENGTH CODE.
SrcPt	TRANSPORT SOURCE PORT.
DstPt	TRASNPORT DESTINATION PORT.
SrcAd	NETWORK SOURCE ADDRESS.
DstAd	NETWORK DESTINATION ADDRESS.
EnbCd	RECEIVE ENABLE CODE (2-BITS).
DblCd	RECEIVE DISABLE CODE.
ArmCd	RECEIVE ARM CODE (SENT FROM SKT ENG TO RCV MGR).
ClrCd	RECEIVE DISARM CODE.
PshCd	RECEIVE PACKET RETURN CODE.
SlwCd	RECEIVE SLOW PATH CODE.
FstCd	RECEIVE FAST PATH CODE.
ReqCd	REQUEST CODE (TO TELL RCVMGR TO PUT HEADERS INTO HEADER BUFFER).
EvtEn	RECEIVE EVENT ENABLE BIT (STORED IN MGR BUF).
SkTpl	SOCKET TUPLE.
RcvPk	RECEIVE PACKET.
PrsHd	PARSE HEADER (GENERATED BY RECEIVE PARSER).
SktDsc	SOCKET DESCRIPTOR.
EvtCd	EVENT CODE (FOR FstEvtQ IT CAN EITHER BE FstCd OR ClrCd). (FOR SlwEvtQ IT CAN EnbCd, DblCd, or SlwCd).

FIG. 5

MNEMONIC	FORMAT
SkTpl	{ XmtAckNum, XmtSeqNum, XmtSeqLmt, XmtCcwSz, MaxSegSz, MaxXmtWin, RcvSeqLmt, ExpRcvSeq, ExpHdrLen, TcbAd }
HdrId	{ EngCx, HdSel }
TplUpd	{ PktAckNum, NxtSeqMax, NxtCcwSz, NxtExpSeq }
RcvPk	{ PktHd, PayLd }
PktHd	{ LnkHd, SnpHd, VlnHd, NetHd, TptHd }
PrsHd	{ PktBufId, HdrCd, PktPaySz, NetIx, TptIx, DdpIx, PkEnd, PrsSt * }
SktDsc	{ HdrCd, SrcPt, DstPt, SrcAd, DstAd, DetEn }
PkEnd	{ BufId, PkLen }
NetHd	{ PktSrcAdr, PktDstAdr, * }
TptHd	{ PktRcvSeq, PktXmtAck, PktXmtWin, PktSrcPrt, PktDstPrt, * }

* = INCLUDES OTHER VARIABLES NOT LISTED

FIG. 6

```
(700)  RcvMac
(701)  parse   RcvPk
(702)  write   RcvMacQ            { RcvPk                        }      // FORWARD RECEIVE PACKET.
(703)  write   RcvMacQ            { RcvSt                        }      // APPEND STATUS.

(704)  RcvPrs
(705)  read    RcvBufQ            { BufId                        }      // GET A RECEIVE BUFFER.
(706)  parse   RcvMacQ            { RcvPk    , RcvSt             }      // PARSE A PACKET.
(707)  write   RcvBuf     [BufId] { PrsHd    , RcvPk             }      // SAVE PACKET AND PARSE INFO.
(708)  write   PrsEvtQ            { PkEnd    , SkHsh   , SktDsc  }      // SEND EVENT DESCRIPTOR.

(709)  SktDet
(710)  read    PrsEvtQ            { PkEnd    , SkHsh   , SktDsc  }      // GET EVENT DESCRIPTOR.
(711)  test    SktDscBuf [SkHsh]  { SktDsc   , .....     SktDsc  } , SktDsc  // SEE IF FAST PATH.
(712)  write   DetEvtQ            { SlwCd    , HdrCd   , PkEnd   }      // SEND SLOW DESCRIPTOR.

(713)  RcvMgr
(714)  read    DetEvtQ            { SlwCd    , HdrCd   , PkEnd   }      // GET SLOW DESCRIPTOR.
(715)  write   SlwEvtQ            { SlwCd    , HdrCd   , PkEnd   }      // PASS TO SktEng.

(716)  SktEng
(717)  read    SlwEvtQ            { SlwCd    , HdrCd   , PkEnd   }      // GET SLOW DESCRIPTOR.
(718)  read    MsgBufQ            { MsgAd                        }      // GET HOST BUFFER.
(719)  copy    RcvBuf     [BufId] { PrsHd    , RcvPk             } , HstMem[MsgAd]  // COPY PACKET.
(720)  write   HstMem     [MsgAd] { MsgHd                        }      // NOTIFY HOST.
(721)  write   RcvBufQ            { BufId                        }      // RECYCLE RECEIVE BUFFER.
```

SLOW PATH RECEIVE SEQUENCE

FIG. 7

```
      FIRST PHASE (SLOW-PATH PURGE)

(800) Host
(801) write    HstMem  [CmdAd] { CmdHd  , TcbAd  }          // WRITE RECEIVE COMMAND.
(802) write    HstEvtQ         { CmdAd  , TcbId  }          // SEND COMMAND NOTICE.


(803) SktEng
(804) read     HstEvtQ         { CmdAd  , TcbId  }          // GET NOTICE COMMAND.
(805) read     HstMem  [CmdAd] { CmdHd  , TcbAd  }          // LOAD EnbSk COMMAND.
(806) copy     HstMem  [TcbAd] { SktDsc          } , SktDscBuf [TcbId] // LOAD SOCKET DESCRIPTOR.
(807) write    DetCmdQ         { EnbCd  , TcbId  }          // SEND DETECT ENABLE COMMAND.

(808) SktDet
(809) read     DetCmdQ         { EnbCd  , TcbId  }          // GET DETECT ENABLE COMMAND.
(810) set      SktDscBuf [TcbId] { DetEn          }          // ENABLE SOCKET DETECTION.
(811) write    DetEvtQ         { EnbCd  , TcbId  }          // SEND PURGE MARKER.

(812) RcvMgr
(813) read     DetEvtQ         { EnbCd  , TcbId  }          // GET PURGE MARKER.
(814) write    SlwEvtQ         { EnbCd  , TcbId  }          // SEND TO SOCKET ENGINE.

(815) SktEng
(816) read     SlwEvtQ         { EnbCd  , TcbId  }          // GET PURGE MARKER.
(817) read     MsgBufQ         { MsgAd           }          // GET A HOST MESSAGE BUFFER.
(818) write    HstMem  [MsgAd] { MsgHd           }          // NOTIFY HOST OF COMMAND DONE.
                                                               (THE MsgHd INDICATES AN ENABLE
                                                               MARK MESSAGE EnbMrkMsg.)

      SECOND PHASE (LOAD SOCKET STATE)

(819) Host
(820) read     HstMem  [MsgAd] { MsgHd           }    // GET MESSAGE THAT PRIOR COMMAND IS DONE.
(821) write    HstMem  [TcbAd] { SkTpl           }    // WRITE SOCKET STATE.
(822) write    HstMem  [CmdAd] { CmdHd  , TcbAd  }    // WRITE RECEIVE COMMAND.
(823) write    HstEvtQ         { CmdAd  , TcbId  }    // SEND COMMAND NOTICE.


(824) SktEng
(825) read     HstEvtQ         { CmdAd  , TcbId  }    // GET NOTICE COMMAND.
(826) read     HstMem  [CmdAd] { CmdHd  , TcbAd  }    // LOAD HOST COMMAND.
(827) copy     HstMem  [TcbAd] { SkTpl           } ,TcbBuf [TcbId]    // MOVE SOCKET STATE TO NID.
(828) write    MgrCmdQ         { ArmCd  , TcbId  }    // SEND ARM COMMAND TO RCV MGR.

(829) RcvMgr
(830) read     MgrCmdQ         { ArmCd  , TcbId  }    // GET ARM COMMAND.
(831) if       (SktRcvQRdy[TcbId])
(832)   write  FstEvtQ [TcbAd] { FstCd  , TcbId  }    // PUT FAST-PATH EVENT TO SKT ENG.
(833) else
(834)   set    MgrBuf  [TcbId] { EvtEn           }    // SET EVENT ENABLE BIT SO NEXT
                                                         TIME A FAST-PATH DESCRIPTOR
                                                         APPREARS ON SKT RCV QUEUE, IT
                                                         WILL BE PASSED TO SOCKET
                                                         ENGINE AS A FAST-PATH EVENT.
```

CONNECTION HANDOUT SEQUENCE

FIG. 8

```
(900) RcvMac
(901) parse  RcvPk
(902) write  RcvMacQ                 { RcvPk                     }      // FORWARD RECEIVE PACKET.
(903) write  RcvMacQ                 { RcvSt                     }      // APPEND STATUS.

(904) RcvPrs
(905) read   RcvBufQ                 { BufId                     }      // GET A RECEIVE BUFFER.
(906) parse  RcvMacQ                 { RcvPk  , RcvSt            }      // PARSE A PACKET.
(907) write  RcvBuf     [BufId]      { PrsHd  , RcvPk            }      // SAVE PACKET AND PARSE INFO.
(908) write  PrsEvtQ                 { PkEnd  , SkHsh , SktDsc   }      // SEND EVENT DESCRIPTOR.

(909) SktDet
(910) read   PrsEvtQ                 { PkEnd  , SkHsh , SktDsc   }            // GET EVENT DESCRIPTOR.
(911) test   SktDscBuf [SkHsh]       { SktDsc , ....  , SktDsc   } , SktDsc   // SEE IF FAST-PATH FRAME.
(912) write  DetEvtQ                 { FstCd  , HdrCd , PkEnd    }            // SEND FAST DESCRIPTOR.
(913) write  DetEvtQ                 { TcbId  ,                  }

(914) RcvMgr
(915) read   DetEvtQ                 { FstCd  , HdrCd , PkEnd    }      // GET FAST DESCRIPTOR.
(916) read   DetEvtQ                 { TcbId                     }
(917) write  SktRcvQ    [TcbId]      { SlwCd  , HdrCd , PkEnd    }      // SAVE TO SOCKET QUEUE.
(918) if     (MgrBuf[TcbId]{EvtEn})  begin                       }
(919) write  FstEvtQ                 { FstCd  , TcbId            }      // SEND FAST EVENT NOTICE.
(920) clr    MgrBuf     [TcbId]      { EvtEn                     }      // CLR EVENT ENABLE.
(921) end

(922) SktEng
(923) read   FstEvtQ                 { FstCd  , TcbId            }      // GET FAST EVENT NOTICE.
(924) write  MgrCmdQ                 { ReqCd  , HdrId , TcbId    }      // REQUEST HEADER DELIVERY.

(925) RcvMgr
(926) read   MgrCmdQ                 { ReqCd  , HdrId , TcbId    }      // GET HEADER REQUEST.
(927) read   SktRcvQ    [TcbId]      { SlwCd  , HdrCd , PkEnd    }      // GET RCV DESCRIPTOR
(928) copy   RcvBuf     [BufId]      { PrsHd  , PktHd            }, HdrBuf [HdrId] // GET FAST HEADERS.
(929) write  HdrEvtQ                 { TcbId                     }      // SEND HEADER EVENT.

(930) SktEng
(931) read   HdrEvtQ                 { TcbId                     }      // GET HEADER EVENT.

(932) Check  packet ack win and seq;
(933) test   HdrBuf     [HdrId]      against  TcbBuf[TcbId]             // CHECK ACK, WINDOW AND SEQ.

(934) if (TcbBuf [TcbId]{MdlVd}) begin                                 // IF VALID MEM DESCRIPTOR.
(935)    read    HdrBuf    [HdrId]   { BufId   }                        // GET SOURCE POINTER.
(936)    read    TcbBuf    [TcbId]   { ApDsc   }                        // GET DESTINATION POINTER.
(937)    copy    RcvBuf    [BufId]   { PayLd   }    , HstMem[ApDsc]     //  MOVE FAST DATA.
(938)    read    MsgBufQ             { MsgAd   }                        // GET A HOST BUFFER.
(939)    write   HstMem   [MsgAD]    { MsgHd   }                        // SEND RESPONSE MESSAGE.
(940)    write   RcvBufQ             { BufId   }                        // RECYCLE MDL VALID BIT.
(941)    clear   TcbBuf    [TcbId]   { MdlVd   }                        // CLEAR MDL VALID BIT.
(942)    end
```

FAST-PATH RECEIVE SEQUENCE

FIG. 9A

```
(943) else  begin                                   // IF NO VALID MEMORY DESCRIPTOR, SEND INITIAL DATA.
(944)    read    MsgBufQ          { MsgAd  }                      // GET A HOST BUFFER.
(945)    copy    RcvBuf   [BufId] { PayLd  }                      // COPY FAST DATA.
(946)    write   HstMem   [MsgAd] { MsgHd  }  , HstMem[MsgAd]     // REQUEST MEMORY DESCRIPTOR.
(947)    write   RcvBufQ          { BufId  }                      // RECYCLE RECEIVE BUFFER.

(948) Host
(949) read    HstMem   [MsgAd] { MsgHd  , MsgDt  }                // GET RECEIVE REQUEST.
(950) write   HstMem   [CmdAd] { CmdHd  , ApDsc  }                // WRITE RECEIVE COMMAND.
(951) write   HstEvtQ          { CmdAd  , TcbId  }                // SEND COMMAND NOTICE.

(952) SktEng
(953) read    HstEvtQ          {CmdAd , TcbId }                   // GET COMMAND NOTICE.
(954) copy    HstMem   [CmdAd] {CmdHd , ApDsc } , TcbBuf [TcbId]  // LOAD RECEIVE COMMAND.
(955) set     TcbBuf   [TcbId] { MdlVd ,      }                   // SET VALID BIT.
(956) write   MgrCmdQ          { ArmCd , TcbId }                  // LOAD RECEIVE COMMAND.
                                               }
(957) RcvMgr
(958) read    MgrCmdQ          { ArmCd, TcbId  }                  // GET ARM COMMAND.
(959) if (SktRcvQRdy [TcbId])
(960)   write FstEvtQ          { FstCd , TcbId  }                 // SEND NOTICE TO SKT ENG.
(961) end
```

FAST-PATH RECEIVE SEQUENCE (CONTINUED)

FIG. 9B

KEY TO FIG. 9

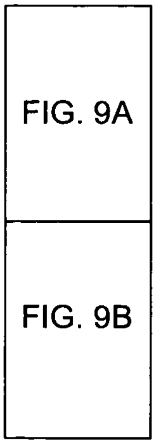


FIG. 9

```
Idle: if (SlwEvtQRdy) begin                        // SERVICE SLOW EVENT QUEUE.
        if (SlwEvtQ {EvtCd} = = 0) begin           // SLOW RECEIVE EVENT.
          EState    <=  SlwRcvEvt;                 // GO TO EVENT SERVICE.
          ETcbId    <=  SlwEvtQ{TcbId};            // SAVE TCB NUMBER.
          ECmdAd    <=  x;                         //
          EHdrCd    <=  SlwEvtQ{HdrCd};            // SAVE HEADER DMA LENGTH.
          EHdrAd    <=  x;                         //
          EBufId    <=  SlwEvtQ{BufId};            // SAVE RECEIVE BUFFER NUMBER.
          EPkLen    <=  SlwEvtQ{PkLen};            // SAVE RECEIVE BUFFER LENGTH.
        end

        else if (SlwEvtQ {EvtCd} = = 1) begin      // SLOW MARK EVENT.
          EState    <=  EnbMrkEvt;                 // GO TO EVENT SERVICE.
          ETcbId    <=  SlwEvtQ{TcbId};            // SAVE TCB NUMBER.
          EcmdAd    <=  x;                         //
          EHdrCd    <=  x;                         //
          EHdrAd    <=  x;                         //
          EBufId    <=  x;                         //
          EPkLen    <=  x;                         //
        end

        else begin                                 // DISABLE MARK EVENT.
          EState    <=  DblMrkEvt;                 // GO TO EVENT SERVICE.
          ETcbId    <=  SlwEvtQ{TcbId};            // SAVE TCB NUMBER.
          EcmdAd    <=  x;                         //
          EHdrCd    <=  SlwEvtQ{HdrCd};            // SAVE HEADER DMA LENGTH.
          EHdrAd    <=  x;                         //
          EBufId    <=  SlwEvtQ{BufId};            // SAVE RECEIVE BUFFER NUMBER.
          EPkLen    <=  SlwEvtQ{PkLen};            // SAVE RECEIVE BUFFER LENGTH.
        end
      end

      else if (FstEvtQRdy) begin                 // SERVICE FAST EVENT QUEUE.
        if (FstEvtQ {EvtCd} = = 0) begin         // FAST RECEIVE EVENT (EvtCd is the FstCd).
          EState    <=  FstRcvEvt;               // GO TO EVENT SERVICE.
          ETcbId    <=  FstEvtQ{TcbId};          // SAVE TCB NUMBER.
          EcmdAd    <=  x;                       //
          EHdrCd    <=  x;                       //
          EHdrAd    <=  FstEvtQ{HdrId};          // SAVE HEADER BUFFER POINTER.
          EBufId    <=  x;                       //
          EPkLen    <=  x;                       //
        end
        else begin                               // CLEAR MARK EVENT (EvtCd is the ClrCd).
          EState    <=  ClrMrkEvt;               // GO TO EVENT SERVICE.
          ETcbId    <=  FstEvtQ{TcbId};          // SAVE TCB NUMBER.
          EcmdAd    <=  x;                       //
          EHdrCd    <=  x;                       //
          EHdrAd    <=  FstEvtQ{HdrId};          // SAVE HEADER BUFFER POINTER.
          EBufId    <=  x;                       //
          EPkLen    <=  x;                       //
        end
      end
```

SOCKET ENGINE STATES

FIG. 10A

```
else if (HstEvtQRdy) begin                              // SERVICE HOST EVENT QUEUE.
    EState   <= SktCmdEvt;                              // GO TO EVENT SERVICE.
    ETcbId   <= HstEvtQ{TcbId};                         // SAVE TCB NUMBER.
    EcmdAd   <= HstEvtQ{CmdAd};                         // SAVE COMMAND BLOCK ADDRESS.
    EHdrCd   <= x;                                      //
    EHdrAd   <= x;                                      //
    EBufId   <= x;                                      //
    EPkLen   <= x;                                      //
   end

else if (HdrEvtQRdy) begin              // SERVICE HEADER EVENT QUEUE.
    EState   <= HdrDmaEvt;              // GO TO EVENT SERVICE.
    ETcbId   <= HdrEvtQ{TcbId};         // SAVE TCB NUMBER IN ETcbId (MAKES
    EcmdAd   <= x;                         ALL THE BITS OF THE TCB SIMULTANEOUSLY
    EHdrCd   <= x;                         AVAILABLE TO THE SKT ENG.)
    EHdrAd   <= HdrEvtQ{HdrId};         // SAVE HEADER BUFFER POINTER IN EHdrAd
    EBufId   <= x;                         (MAKES ALL THE BITS OF THE HEADER BUFFER
    EPkLen   <= x;                         SIMULTANEOUSLY AVAILABLE TO THE SKT ENG.)
   end

  else begin                                            // NO EVENT TO SERVICE.
    EState   <= Idle;                                   // KEEP CHECKING FOR WORK.
    ETcbId   <= HdrEvtQ{TcbId};                         //
    EcmdAd   <= x;                                      //
    EHdrCd   <= x;                                      //
    EHdrAd   <= x;                                      //
    EBufId   <= x;                                      //
    EPkLen   <= x;                                      //
   end

// Slow Path Receive Event.
SlwRcvEvt: begin                                        // DMA SLOW-PATH PACKET TO HOST.
    EState      <= SlwRcv0;                             // SET NEXT STATE.
    EMsgAd      <= MsgBufQ{MsgAd};                      // GET HOST BUFFER.
    DrmAd       <= EBufId<<11;                          // DRAM SOURCE ADDRESS.
    HstAd       <= MsgBufQ{MsgAd} + MsgHdLen;   // HOST DESTINATION ADDRESS.
    HstSz       <= EPkLen;                              // DMA LENGTH.
    HstDmaCmd   <= R2hCd;                               // MOVE RCV BUFFER TO HOST DMA.
   end

SlwRcv0: begin                                          // SEND HOST NOTIFICATION MESSAGE.
    EState      <= Idle;                                // GO FIND WORK.
    E2hBuf      <= SlwRcvMsg;                           // SET UP SLOW RECEIVE MESSAGE.
    HstAd       <= EMsgAd;                              // HOST DESTINATION ADDRESS.
    HstSz       <= MsgHdLen;                            // DMA LENGTH.
    HstDmaCmd   <= E2hCd;                               // MOVE MESSAGE TO HOST.
    RcvBufQ     <= EBufId;                              // RECYCLE RECEIVE BUFFER.
   end
```

SOCKET ENGINE STATES (CONTINUED)

FIG. 10B

```
// Slow-Path Purge Event (Socket Detection Enabled).
  EnbMrkEvt: begin                          // NOTIFY HOST FIRST PHASE OF HANDOUT IS DONE.
      EState      <= Idle;                  // GO FIND WORK.
      EMsgAd      <= MsgBufQ{MsgAd};        // GET HOST MESSAGE BUFFER ADDRESS.
      E2hBuf      <= EnbMrkMsg;             // PREPARE AN ENABLE-MARK MESSAGE.
      HstAd       <= MsgBufQ{MsgAd};        // SET HOST DESTINATION ADDRESS.
      HstSz       <= MsgHdLen;              // SET DMA LENGTH.
      HstDmaCmd   <= E2hCd;                 // MOVE MESSAGE TO HOST MESSAGE BUFFER.
     end

// Descriptor Buffer Release Event (Socket Detection Disabled).
  DblMrkEvt: begin                                // DISABLE MARK EVENT.
      EState      <= DblMrk0;                     // GO TO DISABLE SERVICE.
      EMsgAd      <= MsgBufQ{MsgAd};              // GET HOST BUFFER.
     end

  DblMrk0: begin                                  // NOTIFY HOST.
      EState      <= Idle;                        // GO FIND WORK.
      E2hBuf      <= DblMrkMsg;                   // DISABLE-MARK MESSAGE.
      HstAd       <= EMsgAd;                      // HOST DESTINATION ADDRESS.
      HstSz       <= MsgHdLen;                    // DMA LENGTH.
      HstDmaCmd   <= E2hCd                        // DO NID TO HOST DMA.
     end

// Fast-Path Receive Event.
  FstRcvEvt: begin                                // GET FAST PACKET HEADER.
      EState      <= Idle;                        // GO FIND WORK.
      MgrCmdQ     <= {ReqCd, PktId, ETcbId}       // REQUEST HEADER DELIVERY.
     end

// Fast-Receive Purge Event (Fast Event Disabled).
  ClrMrkEvt: begin
      EState      <= ClrMrk0;                  // GO TO NEXT STATE.
      EMsgAd      <= MsgBufQ{MsgAd};           // GET MESSAGE BUFFER ON HOST.
      HstTcbId    <= ETcbId;                   // SOURCE IS TCB BUFFER.
      HstAd       <= TcbBuf{TcbAd};            // DESTINATION ON HOST.
      HstSz       <= SkTplLen;                 // DMA LENGTH.
      HstDmaCmd   <= T2hCd;                    // MOVE TCB FROM NID TO HOST.
      DetCmdQ     <= {DblCd, ETcbId};          // SEND DISABLE COMMAND TO SKT DET.
     end

  ClrMrk0:  begin                              // NOTIFY HOST THAT STATE HAS BEEN EXPORTED.
      EState      <= Idle;                     // GO FIND WORK.
      E2hBuf      <= ExportMsg;                // STATE EXPORT MESSAGE INTO E2HBUF.
      HstAd       <= EMsgAd;                   // DESTINATION ADDRESS ON HOST.
      HstSz       <= MsgHdLen;                 // DMA LENGTH.
      HstDmaCmd   <= E2hCd;                    // MOVE MESSAGE FROM NID TO HOST.
     end

// Host Command Entry Event.
  SktCmdEvt: begin                             // GET COMMAND FROM HOST.
      EState      <= SktCmd0;                  // SET NEXT STATE.
      HstAd       <= ECmdAd;                   // SET HOST SOURCE ADDRESS.
      HstSz       <= CmdHdLen;                 // SET DMA LENGTH.
      HstDma Cmd  <= H2eCd;                    // MOVE COMMAND FROM HOST TO NID.
     end
```

SOCKET ENGINE STATES (CONTINUED)

FIG. 10C

```
SktCmd0:  if (H2eBuf {CmdCd} = = 0) begin       // IF ENABLE COMMAND.
    EState          <=  SktEnbCmd;              // GO TO ENABLE ROUTINE.
   end

   else if (H2eBuf {CmdCd} = = 1) begin         // IF ARM COMMAND.
    EState          <=  SktArmCmd;              // GO TO ARM ROUTINE.
   end

   else begin                                   // MUST BE RCV COMMAND.
    EState          <=  SktRcvCmd;              // GO TO ARM ROUTINE.
   end

// Socket Enable Command Service.
  SktEnbCmd: begin                               // GET SOCKET DESCRIPTOR.
      EState        <=  SktEnb0;                 // GO TO NEXT STATE.
      DscBufAd      <=  ETcbId * SktDscLen;      // ADDR FOR SktDsc BUFFER.
      HstAd         <=  H2eBuf {TcbAd} + SktDscIx; // HOST SktDsc ADDRESS.
      HstSz         <=  SktDscLen;               // DMA LENGTH.
      HstDmaCmd <=  H2dCd;                       // MOVE SkDsc FROM HOST TO NID.
     end

  SktEnb0: begin                                 // ENABLE SOCKET DETECTION.
      EState        <=  Idle;                    // GO FIND WORK.
      DetCmdQ       <=  {EnbCd, ETcbId};         // SEND ENABLE COMMAND.
     end

// Socket Arm Command Service.
  SktArmCmd: begin                               // GET SOCKET STATE.
      EState        <=  SktArm0;                 // GO TO NEXT STATE.
      TcbBufAd      <=  ETcbId * TcbBufLen;      // ADDR FOR SktDsc BUFFER.
      HstAd         <=  H2eBuf {TcbAd} + SkTplIx;// HOST SktTpl ADDRESS.
      HstSz         <=  SkTplLen;                // DMA LENGTH.
      HstDmaCmd <=  H2tCd;                       // MOVE SkTpl FROM HOST TO NID.
     end

  SktArm0: begin                                 // ENABLE SOCKET RECEIVE.
      EState        <=  Idle;                    // GO FIND WORK.
      MgrCmdQ       <=  ArmCd, ETcbId};          // SEND ARM RECEIVE COMMAND.
     end

// Socket Receive Command Service.
  SktRcvCmd: begin                               // GET APPLICATION BUFFER DSC.
      EState        <=  SktRcv0;                 // GO TO NEXT STATE.
      TcbBufAd      <= (ETcbId * TcbBufLen) + ApDscIx;  // ADDR FOR APP DSC BUFFER.
      HstAd         <=  ECmdAd + ApDscIx;           // HOST ApDsc ADDRESS.
      HstSz         <=  ApDscLen;                // DMA LENGTH.
      HstDmaCmd <=  H2tCd;                       // MOVE ApDsc FROM HOST TO NID.
     end

  SktRcv0: begin                                    // ENABLE SOCKET RECEIVE.
      EState            <=  Idle;                   // GO FIND WORK.
      TcbBuf{RSqMx} <=  TcbBuf{RSqMx}+H2eBuf{SqInc};  // INCREMENT RECEIVE WINDOW.
      MgrCmdQ           <=  {ArmCd, ETcbId};        // SEND ARM RECEIVE COMMAND.
      MdlVd[ETcbId]  <=  1;                         // ADDR FOR MDL VALID BIT.
     end
```

SOCKET ENGINE STATES (CONTINUED)

FIG. 10D

```
// Header Event Service.
  HdrDmaEvt:  begin
              EState      <= HdrEvt0;                // GO TO NEXT STATE.
              EMsgAd      <= MsgBufQ {MsgAd};        // GET HOST BUFFER.
              EFlush      <= FlushDet;           // FLUSH DETECT - PARALLEL OPERATION
              EBufId      <= HdrBuf{BufId};          // SAVE RECEIVE BUFFER NUMBER.
              EPkLen      <= HdrBufQ{PkLen};         // SAVE PACKET LENGTH.
            end

  HdrEvt0:  if (EFlush) begin                        // IF SOCKET RECEIVE ERROR.
              EState      <= Idle;                   // GO FIND WORK.
              MgrCmdQ     <=   {PshCd, ETcbId};      // RETURN PACKET AND FLUSH.
            end

            else if (MdlVd[ETcbId]) begin            // IF APP BUFFER DSCR IS VALID.
              EState      <= FastRcv;                // GO TO NEXT STATE.
              DrmAd       <= (EBufId<<11)+PkHdrLen;  // DRAM SOURCE ADDRESS.
              HstAd       <= TcbBuf{ApDsc};          // HOST DESTINATION ADDRESS.
              HstSz       <= EPkLen - HdrLen;        // DMA LENGTH.
              HstDmaCmd   <= R2hCd;                  // DO RCV TO HOST DMA.
            end

            else                                     // SEND FIRST FAST-PATH PACKET TO HOST.
              EState      <= InitRcv;                // SET NEXT STATE.
              DrmAd       <= EBufId<<11;             // SET DRAM SOURCE ADDRESS.
              HstAd       <= EMsgAd + MsgHdLen;      // SET HOST DESTINATION ADDRESS.
              HstSz       <= EPkLen;                 // SET DMA LENGTH.
              HstDmaCmd   <= R2hCd;                  // MOVE PACKET TO HOST.
              RcvBufQ     <= EBufId;                 // RECYCLE RECEIVE BUFFER.
            end
   FastRcv:  begin                             // NOTIFY HOST.
              EState      <= UpdMdlEntries;    //
              E2hBuf      <= FstRcvMsg;        // SEND FAST-PATH RECEIVE MESSAGE.
              HstAd       <= EMsgAd;           // SET HOST DESTINATION ADDRESS.
              HstSz       <= MsgHdLen;         // SET DMA LENGTH.
              HstDmaCmd   <= E2hCd;            // MOVE MESSAGE TO HOST.
              RcvBufQ     <= EBufId;           // RECYCLE RECEIVE BUFFER.
              TcbBuf      <= TplUpd;     // UPDATE SOCKET STATE - PARALLEL OPERATION.
              MgrCmdQ     <= {ArmCd, ETcbId};  // SEND ARM RECEIVE COMMAND TO RCV MGR.
            end
   UpdMdlEntries:
            if exhausted begin                       // IF THE MDL ENTRY IS EXHAUSTED
              clear TcbBuf[TcbId]{MdlVd};            // CLEAR THE MDL VALID BIT
            end
              EState      <= FastRcv;                // GO TO THE IDLE STATE

   InitRcv:  begin     // NOTIFY HOST THAT FIRST FAST-PATH PACKET IS IN MESSAGE BUFFER.
              EState      <= Idle;                   // GO FIND WORK.
              E2hBuf      <= RcvReqMsg;              // FORM RECEIVE REQUEST MESSAGE.
              HstAd       <= EMsgAd;                 // SET HOST DESTINATION ADDRESS.
              HstSz       <= MsgHdLen;               // SET DMA LENGTH.
              HstDmaCmd   <= E2hCd;                  // MOVE MESSAGE TO HOST.
              RcvBufQ     <= EBufId;                 // RECYCLE RECEIVE BUFFER.
              TcbBuf      <= TplUpd;     // UPDATE SOCKET STATE - PARALLEL OPERATION.
            end
```

SOCKET ENGINE STATES (CONTINUED)

FIG. 10E

KEY TO FIG. 10

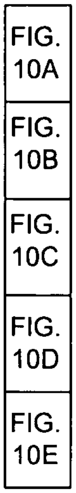
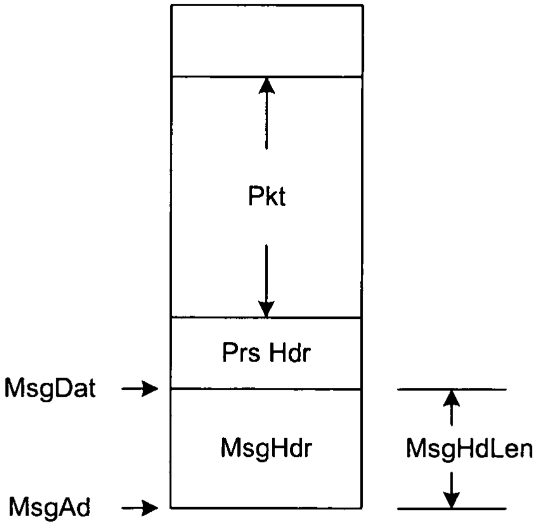


FIG. 10



HOST MESSAGE BUFFER

FIG. 11

```
        FIRST PHASE  (FAST-PATH EVENT PURGE)
(1500)  SktEng
(1501)  write   MgrCmdQ           { PshCd   , TcbId }         // SEND DISARM COMMAND.

(1502)  RcvMgr
(1503)  read    MgrCmdQ           { PshCd   , TcbId }         // GET DISARM COMMAND.
(1504)  write   FstEvtQ           { ClrCd   , TcbId }         // SEND PURGE MARKER.
(1505)  clear   MgrBuf   [TcbId]  { EvtEn }                   // DISARM SOCKET RECEIVE.


        SECOND PHASE (SOCKET STATE SAVE)
(1506)  SktEng
(1507)  read    FstEvtQ           { ClrCd   , TcbId }            // GET DISARM MARKER.
(1508)  copy    TcbBuf   [TcbId]  { SkTpl   } , HstMem[TcbAd]  // SEND STATE TO HOST.
                                                              (ALSO SEE LINES 1123-1126).
(1509)  read    MsgBufQ           { MsgAd   }                  // GET MESSAGE BUFFER.
(1510)  write   HstMem   [MsgAd]  { MsgHd   }                  // NOTIFY HOST.


        THIRD PHASE (FAST-PATH QUEUE PURGE)
(1510)  SktEng
(1511)  write   DetCmdQ           { DblCd   , TcbId }         // SEND DISABLE COMMAND.

(1512)  SktDet
(1513)  read    DetCmdQ           { DblCd   , TcbId }         // GET DISABLE COMMAND.
(1514)  clr    SktDscBuf [TcbId]  { DetEn           }         // DISABLE SOCKET DETECTION.
(1515)  write   DetEvtQ           { DblCd   , TcbId }         // SEND PURGE MARKER.

(1516)  RcvMgr
(1517)  read    DetCmdQ         { DblCd , TcbId    }          // GET PURGE MARKER.
(1518)  while   (SktRcvQRdy[TcbId])                           // IF SOCKET RCV DESCRIPTORS.
(1519)    copy   DetEvtQ    {SlwCd , HdrCd , PkEnd } , SlwEvtQ  // MOVE DESCRIPTOR TO SLOW QUEUE.
(1520)  write   SlwEvtQ         { DblCd , TcbId    }          // SEND PURGE MARKER.

(1521)  SktEng
(1522)  read    SlwEvtQ         { DblCd , TcbId    }          // GET PURGE MARKER.
```

CONNECTION FLUSH SEQUENCE

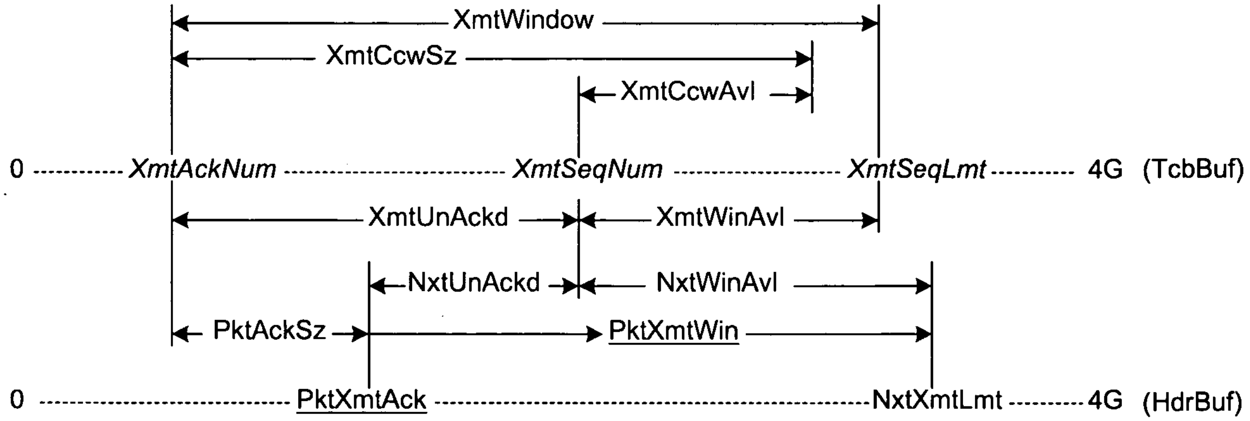
FIG. 12

```
(1600) XmtWindow  = XmtSeqLmt   -  XmtAckNum                       ; // TRANSMIT WINDOW.
(1601) PktAckSz   = PktXmtAck   -  XmtAckNum                       ; // DATA BEING ACKED.
(1602) XmtUnAckd  = XmtSeqNum   -  XmtAckNum                       ; // DATA TO BE ACKED.
(1603) XmtWinAvl  = XmtSeqLmt   -  XmtSeqNum                       ; // AVAILABLE WINDOW.
(1604) CurCcwInc  = XmtCcwSz    +  (MaxSegSz<<1)                   ; // NEW CONGESTION WINDOW.
(1605) NxtUnAckd  = XmtSeqNum   -  PktXmtAck                       ; // DATA TO BE ACKED.
(1606) NxtXmtLmt  = PktXmtAck   +  PckXmtWin                       ; // TRANSMIT LIMIT.
(1607) NxtWinAvl  = NxtXmtLmt   -  XmtSeqNum                       ; // AVAILABLE WINDOW.
(1608) XmtAckNew  = PktXmtAck   != XmtAckNum                       ; // NEW ACK DETECT.
(1609) XmtWinNew  = PktXmtWin   != XmtWindow                       ; // WINDOW CHANGE.

(1610) XmtAckDup  = !XmtAckNew  &  !XmtWinNew                      ; // ACK IS DUPLICATE.
(1611) XmtAckVld  = PktAckSz    <= XmtUnAckd                       ; // ACK IS VALID.
(1612) XmtAckOld  = PktAckSz    >  XmtWindow                       ; // OLD ACK DETECT.
(1613) CurCcwStp  = XmtCcwSz    <  NxtXmtSz                        ; // CONGESTION WINDOW STOP.
(1614) CurWinStp  = XmtWinAvl   <  NxtXmtSz                        ; // WINDOW STOP.
(1615) NxtSlwDet  = CurCcwInc   <  MaxXmtWin                       ; // SLOW START DETECT.
(1616) NxtWinGrw  = PktXmtWin   >  MaxXmtWin                       ; // XMT WINDOW IS GROWING.
(1617) NxtWinOpn  = (XmtSeqLmt  -  NxtXmtLmt)  !<  'Quadrant       ; // XMT WINDOW IS OPENING.
(1618) NxtWinShr  = (NxtXmtLmt  -  XmtSeqLmt)  !<  'Quadrant       ; // XMT WINDOW IS SHRINKING.
(1619) NxtWinStp  = NxtWinAvl   <  NxtXmtSz                        ; // WINDOW STOP.
(1620) NxtXmtCcw  = !XmtAckNew  ?  XmtCcwSz                          // NEXT CONGESTION CTRL WIN.
(1621)            : NxtSlwDet   ?  CurCcwInc    :  MaxXmtWin       ;
(1622) NxtCcwStp  = !XmtAckNew  ?                                  ; // CONGESTION WINDOW STOP.
(1623)            = ((XmtCcwSz  -  NxtUnAckd)   << NxtXmtSz)
(1624)            : NxtSlwDet   ?
(1625)            = ((CurCcwInc -  NxtUnAckd    << NxtXmtSz)
(1626)            : ((MaxXmtWin -  NxtUnAckd    << NxtXmtSz)       ;
(1627) XmtAckEvt  = XmtAckNew   &  (PktXmtAck   == XmtSeqNum);       // ACKING ALL SENT DATA.
(1628) XmtWinEvt  = !NxtWinStp  &  (CurWinStp   |  CurCcwStp)        // TRANSMIT THRESHOLD DETECT.
(1629)            & !NxtCcwStp;
```

ITALICS = INDICATES VALUE FROM TCB BUFFER
UNDERLINE = INDICATES VALUE FROM HEADER BUFFER

INCOMING ACK PROCESSING

FIG. 13



ITALICS = INDICATES VALUE FROM TCB BUFFER
UNDERLINE = INDICATES VALUE FROM HEADER BUFFER

INCOMING ACK PROCESSING

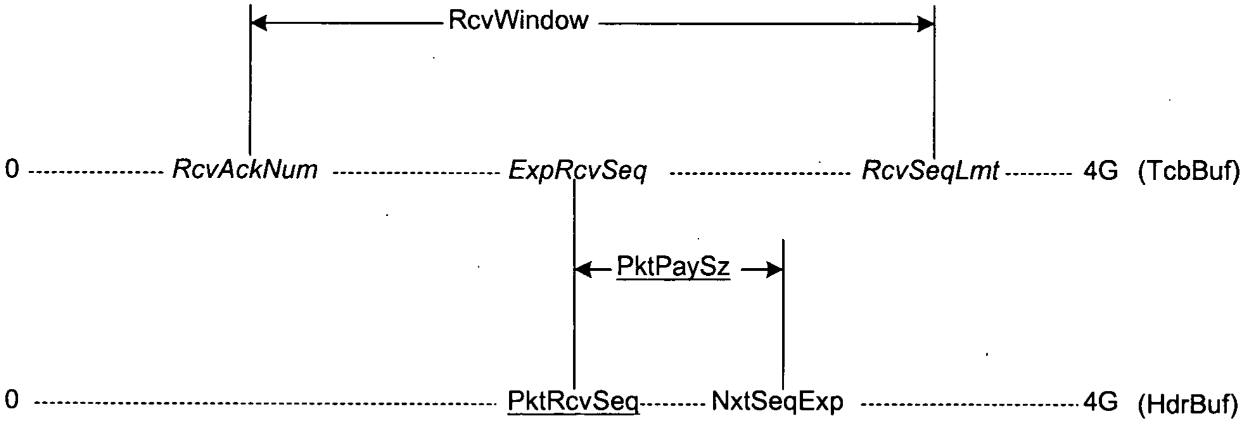
FIG. 14

```
(1700) CurWinFul   = (ExpRcvSeq    == RcvSeqLmt)                    ; //
(1701) HdrLenOk    = (PktHdrLen    == ExpHdrLen)                    ; //
(1702) PurAckDet   = (PktPaySz     == 0)                            ; //

(1703) ExpSeqDet   = (PktRcvSeq    == ExpRcvSeq);                   ; //

(1704) OldSeqDet   = ((RcvSeqLmt   -  PktRcvSeq)  !<  'Quadrant       //
(1705)             | (PktRcvSeq    -  ExpRcvSeq)  !<  'Quadrant     ; //

(1706) WinPrbDet   = (CurWinFul    &  PktPaySz= =1)                 ; //
(1707) NxtSeqExp   = (PktRcvSeq    +  PktPaySz)                     ; //

(1708) CurWinOvr   = ((RcvSeqLmt   -  NxtSeqExp)  !<  'Quadrant)    ; //
(1709) NewDatDet   = ((NxtSeqExp   -  ExpRcvSeq)  <<  'Quadrant)      //
(1710)             &  !OldSeqDet                                    ;
(1711) CurPktFul   = (PktPaySz     == MaxSegSz)                     ; //

(1712) FlushDet    = CurWinOvr    |  (!ExpSeqDet   & !OldSeqDet)      //
(1713)             | NxtWinShr    |  (!XmtAckVld   & !XmtAckOld)    ;
```

ITALICS = INDICATES VALUE FROM TCB BUFFER
UNDERLINE = INDICATES VALUE FROM HEADER BUFFER

INCOMING DATA PROCESSING

FIG. 15



ITALICS = INDICATES VALUE FROM TCB BUFFER
UNDERLINE = INDICATES VALUE FROM HEADER BUFFER

INCOMING DATA PROCESSING

FIG. 16

0 7	8 15	16 23	24 31
FRAME STATUS A	FRAME STATUS A	FRAME STATUS A	FRAME STATUS A
FRAME STATUS B	FRAME STATUS B	FRAME STATUS B	FRAME STATUS B
TIME STAMP	TIME STAMP	TIME STAMP	TIME STAMP
TIME ECHO	TIME ECHO	TIME ECHO	TIME ECHO
SEQUENCE	SEQUENCE	SEQUENCE	SEQUENCE
ACK	ACK	ACK	ACK
WINDOW	WINDOW	PAYLOAD START	PAYLOAD START
PAYLOAD LENGTH	PAYLOAD LENGTH	TCP CHECKSUM	TCP CHECKSUM

HEADER BUFFER FORMAT

THE HEADER BUFFER IS REPRESENTED HERE AS MULTIPLE 32-BIT VALUES TO MAKE THE ILLUSTRATION MORE COMPACT FOR ILLUSTRATIVE PURPOSES. IN ACTUALITY, THE 32-BIT VALUES SET FORTH ABOVE ARE CONCATENATED END-TO-END. THE HEADER BUFFER IS ONE-BIT DEEP AND 8X32 BITS LONG.

FIG. 17

BIT	NAME	DESCRIPTION
31	802.3Oflw	The 802.3 size/count exhausted before the end of the frame.
30	TprtFlags	The transport flags require attention.
29	TprtOpt	Transport header options were detected.
28	TprtOflw	Transport layer completed before the end of the network layer.
27	NetOpt	Network header options were detected.
26	OffsetDet	A nonzero offset value was detected for the network layer.
25	FragDet	Transport fragmentation was detected at the network layer.
24	NetOflw	Network layer completed before the end of the frame.
23	Attn	Attention Bit: Indicates that one of the following conditions occured: !MacAddrDet or IpMcst or MacMcst or !TcpIp or !TcpVer4 or 802.3Uflw or RcvEarly or BufOflw or InvalidPreamble or FcsError or DribbleNibble or CodeViolation or TprtChkErr or TprtHdrLenErr or NetHdrChkErr or NetHdrLenErr or NetUflw.
22	IpBcst	The RcvSeq detected an IP broadcast address.
21	IpMcst	The RcvSeq detected and IP multicast address.
20	PauseDet	The received control frame contained a pause command.
19	CtrlFrame	A control frame was received at the special multicast address.
18	MagicDet	A magic wake up frame was detected.
17	MacBcst	The Mac detected a broadcast destination address.
16	MacMcst	The Mac detected a multicast destination address.
15	MacBDet	Frame's destination address matched the contents of MacAddrB.
14	MacADet	Frame's destination address matched the contents of MacAddrA.
13:12	MacId	Id number of the Mac via which this packet was received.
11:09	SessType	The session layer detected by the RcvSeq. 0 - Session is unknown. 1 - Session is NFS/RPC. 2 - Session is FTP-Data. 3 - Session is WWW-HTTP. 4 - Session is NetBios. 5 - Session is reserved. 6 - Session is reserved. 7 - Session is other protocol.
08:06	TprtType	The transport layer detected by the RcvSeq. 0 - Transport is unknown. 1 - Transport is TcpIp or NlspIpx 2 - Transport is UdpIp or RipIpx. 3 - Transport is NetBiosIpx. 4 - Transport is NcpIpx. 5 - Transport is SpxIpx. 6 - Transport is SapIpx. 7 - Transport is other.

FRAME STATUS A

FIG. 18A

BIT	NAME	DESCRIPTION
05:04	NetVer	The network layer version detected by the RcvSeq. 0 - Network version is unknown. 1 - Network version is other. 2 - Network version is 4. 3 - Network version is 6.
03:00	NetType	The combined network and frame layer types detected by the RcvSeq. 0 - Frame type is unknown. 1 - reserved. 2 - Frame is 802.3 type. 3 - Unused code. 4 - Frame is 802.3 non-snap. 5 - Frame is 802.3 with Snap header. 6 - Frame is unrecognized ethernet type. 7 - Frame is unrecognized 802.3-snap type. 8 - Frame is ethernet control type with type field = 0x8808. 9 - Frame is 802.3-Snap control type with type field = 0x8808. A - Frame is IPX1 on ethernet type. B - Frame is IPX1 on 802.3-snap type. C - Frame is IPX2 on ethernet type. D - Frame is IPX2 on 802.3-snap type. E - Frame is IP on ethernet type. F - Frame is IP on 802.3-Sanp type.

FRAME STATUS A (CONTINUED)

FIG. 18B

KEY TO FIG. 18

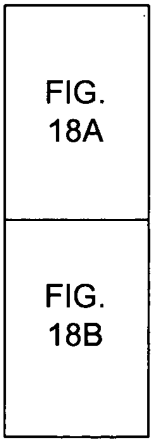


FIG. 18

BIT	NAME	DESCRIPTION
31	802.3Uflw	The frame ended before the 802.3 size/count exhausted.
30	RcvEarly	Data was lost due to insufficient dma bandwidth.
29	BufOflw	The frame length exceeded the capacity of the current buffer.
28	PktMissed	A frame w as missed prior to receiving the current frame.
27	CarrierEvent	Refer to E110 Technical Manual.
26	GoodPacket	Refer to E110 Technical Manual.
25	LongEvent	Refer to E110 Technical Manual.
24	InvldPrmbl	Refer to E110 Technical Manual.
23	CrcErr	Refer to E110 Technical Manual.
22	DrblNbbl	Refer to E110 Technical Manual.
21	CodeErr	Refer to E110 Technical Manual.
20	TprtChkErr	A transport layer checksum error was detected.
19	TprtHdrLen	Transport header length error was detected.
18	NetChkErr	A network header checksum error was detected.
17	NetUflw	The frame ended before the Network length was satisfied.
16	NetHdrLen	Network header length error was detected.
15:08	MacHsh	The cumulative XOR of all bytes of the dest mac address of the packet received.
07:00	CtxHsh	The 8-bit context hash generated by exclusive-oring all bytes of the IP source address, IP destination address, transport source port, and the transport destination port.

FRAME STATUS B

FIG. 19

```
#if (C_code)
/*
 ***************************************************************
 * The constituents of a TCB block are set forth below. The TCB block
 *  is shared between the ATCP driver on the host and the NID. The
 * TCB block is one contiguous block (180 bytes) of fields in the host
 * that gets DMA'd back and forth between host and NID. ULONG is
 * four bytes, UCHAR is one byte, and USHORT is two bytes.
 ***************************************************************
 */
struct tcpcb /* { */

   USHORT  ip_ckbase;                  /* IP base checksum */
   USHORT  tcp_ckbase;                 /* TCP base chksum of template hdr
                                          excluding tcp_seq, tcp_ack & tcp_win
                                          and assuming tcp_hl_flgs = ACK,
                                          and including TCP pseudo-hdr with
                                          payload of 20 (std TCP hdrlen)
                                        */

   ULONG   hosttcbaddrl;               /*  This TCB's address in host mem */
   ULONG   hosttcbaddrh;

   /*
    * The following fields are ordered specifically to match sizes and to match
    * the order in which they are read/written.
    */

   ULONG   max_rcvwnd;         /* rcv win established by host (sb_hiwat) */
   ULONG   max_sndwnd;         /* largest win peer has offered */
   USHORT  t_rttmin;           /* minimum rtt allowed */
   USHORT  pst_timer;          /* timer count for current PST */
   USHORT  t_maxseg;           /* maximum segment size */
   UCHAR   t_dupacks;          /* consecutive dup acks recd */
   UCHAR   t_shflags;          /* flags shared between BSD & NID */
   ULONG   t_rtseq;            /* sequence number being timed */
   ULONG   snd_nxt;            /* send next */
   ULONG   snd_max;            /* highest sequence number sent; used to recognize retransmits */
   ULONG   rcv_adv;            /* advertised window */
   ULONG   snd_cwnd;           /* congestion-controlled win */
   USHORT  rtr_timer;          /* timer count for current RTR */
   UCHAR   t_rxtshift;         /* log(2) of persist exp. backoff */
   UCHAR   rcv_scale;          /* window scaling for recv window */
   USHORT  t_rtt;              /* round trip time (bumped per tick) */
   USHORT  t_srtt;             /* smoothed round-trip time */
   ULONG   snd_una;            /* send unacknowledged */
   ULONG   rcv_nxt;            /* receive next */
```

FIG. 20A

```
    ULONG   rcv_wnd;            /* receive window */
    ULONG   snd_wnd;            /* send window */
    USHORT  t_rttvar;           /* variance in round-trip time */
    USHORT  t_idle;             /* inactivity time (hw?)*/
    USHORT  t_rxtcur;           /* current retransmit value */
    UCHAR   t_rttupdated;       /* number of times rtt sampled */
    UCHAR       snd_scale;      /* window scaling for send window */
    ULONG   snd_wl1;            /* window update seg seq nbr */
    ULONG   snd_wl2;            /* window update seg ack nbr */
    ULONG       ts_recent_age;  /* when TS echo last updated */
    ULONG       ts_recent;      /* timestamp echo data */
    ULONG       last_ack_sent;  /* rcv_nxt of last Ack */
} tcp_stvars;                   /* 100 bytes */

/*
 * Header Template
 */
struct xmit_buffer /* { */
    ULONG       reserved1;
    USHORT      byte_count;     /* byte count of frame to be xmitted */
    USHORT      reserved2;
    ULONG       reserved3;
    ULONG       link;           /* link descriptor to next frame */
    struct inic_frame_hdr /* { */
            USHORT  tmplt_len;  /* len of template hdr (incl this) */
            UCHAR   dhost[6];   /* MAC packet starts here */
            UCHAR   shost[6];
            USHORT  type;
            UCHAR   ip_vhl;
            UCHAR   ip_tos;
            USHORT  ip_len;
            USHORT  ip_id;
            USHORT  ip_fragoff;
            UCHAR   ip_ttl;
            UCHAR   ip_prcl;
            USHORT  ip_csum;
            ULONG   srcaddr;
            ULONG   dstaddr;
            USHORT  srcport;
            USHORT  dstport;
            ULONG   tcp_seq;
            ULONG   tcp_ack;
            USHORT  tcp_hl_flgs;
            USHORT  tcp_win;
            USHORT  tcp_csum;
            USHORT  tcp_urg;
            ULONG   tcp_tsopt;  /* timestamp option */
            ULONG   pad;        /* ensures space for VLAN & TS opt */
    } inic_frame_hdr;
} xmit_buffer;                  /* 16 + 64 bytes */
#endif /* C_code */
```

KEY TO FIG. 20

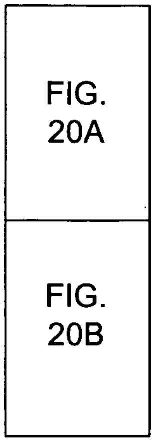


FIG. 20

FIG. 20B